NOTES ON

HITACHI PEACH

DOUBLE DENSITY DOS

BY

PETER CALDER

NOTES ON

HITACHI PEACH DOUBLE DENSITY DOS

Compiled by :-
Peter Calder,
14 Sandy Crescent,
Salisbury Park,
South Australia, 5109.
Phone(a.h.) :- (08)258 0958

The information in these notes has been derived largely from a disassembly of the DOS code. While every attempt has been made to provide correct facts, it is recognized that this has not always been achieved. No responsibility is taken for any loss or damage that may result from the use of the information contained in these notes.

The reader is advised to verify the facts by experimentation before using them where they may cause strife.

The author wishes to acknowledge the patience of his wife. Any reader who is married will know what I mean.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.0.1 The following notes outline the organization of the Double Density Disc Operating System . The details have been obtained from a disassembly of the DOS and in some cases the information may be of doubtful validity. Some details have yet to be fully decyphered, particularly in the case of the File Control Block. The author of these notes would be grateful of any further information from anyone. Some further information may be obtained from "PEACH ROM NOTES" by Bruce Rossell and Howard Viccars,THE SOFTWARE HOUSE,Canberra,ACT.

1.0.2 These notes are not a substitute for the DD DOS manual axailable from HITACHI. It is assumed that the reader has read the manual but seeks to look deeper into the internal workings of the DOS, even if only out of curiosity.

1.0.3 There are a number of differences from Single Density DOS, apart from the fact that sectors are 256 bytes long, there are 40 tracks with 32 sectors/track, and groups consist of 8 sectors.

1.0.4 The directory track is still track 20, but the directory begins on sector 5 and extends to sector 32. As before, file names of 8 characters are allowed,but now a 3 byte extension can be specified to denote file types. The directory now contains the number of bytes used in the last sector of each file , whereas SD DOS had to work it out from the data in the last sector. A maximum of 152 files can be defined. The maximum occurs when each file occupies 1 group only.

1.0.5 Default extensions are :- BAS for BASIC programs,BIN for machine code, DAT for anything else. When BASIC opens a file for output it uses the extension DAT if none is defined in the command.

1.0.6 In double density DOS, the DSKI$ command is now an instruction and not a function. The form of the instruction is now

DSKI$<drive>,<trk>,<sect>,<string>

instead of equating the string to the DSKI$ function.

1.0.7 The DSKI$,DSKO$ tokens for 32K SD DOS are interchanged relative to the DD DOS tokens. It is a good idea to transfer files between SD and DD systems via cassette in 'A' format.

1.0.8 The default device can be changed in the DD DOS by the use of the DEVICE command. This sets $11B to the device number corresponding to the I/O device (02=cassette, $80=drive 0).

1.0.9 DD DOS now provides for the verification of sector writes to the disc. This consists of a read after write to give added confidence when storing data.

1.0.10 Records in DD DOS can now be up to 2048 bytes long and are packed into sectors without any wasted space between records. In SD DOS, each record is stored in its own sector, even if it is only 1 byte long, thus wasting the other 127 bytes.

1.0.11 Most of the bugs in the SD DOS, identified in "PEACH ROM NOTES", seem to have been fixed in DD DOS, and the DOS is now quite robust.

2.0 STRUCTURE OF THE DOS

2.0.1 The DOS is loaded by the DD bootstrap program in the ROM at address $FE7E - $FEFF. This program actually resides on the disc controller card and overlays the bootstrap which is in the BASIC ROM at the same address. The overlay disappears if the disc drive is switched off. The ROM bootstrap is called by the ROM initialization routine (from $FB5E) after the CRTC has been set up. A listing of the bootstrap program is included in APPENDIX I.

2.0.2 The bootstrap loads the DOS loader program from track 0,sectors 1 and 2 of the disc into address $4400. The DOS loader then reads the DOS code from track 1 , starting at sector 1, ending on sector 24. A listing of the DOS loader program is included in APPENDIX II.

2.0.3 The DOS loads into address $8800-$9FFF and begins to execute when loading is completed.

2.1 DOS INITIALIZATION

2.1.1 The first part of the DOS is once-only initialization code which sets up conditions in the low part of RAM (called the communication area),to let ROM BASIC know the DOS is there,and to set up buffers for disc file transfer. This code ends at address $8AFB and is overwritten by BASIC's stack and string area.

2.1.2 The sequence of events that occurs during DOS initialization is as follows:

a) The area used by the DOS loader program ($4400-$4600) is cleared.
b) The PF key names are loaded from $FDDB.
c) Pointers $01CE-$01E0 are given values.
d) COM0 jump table is loaded from $EF39, and buffer space allocated.
e) Pairs of addresses from $FF40 onwards are scanned for response,and further COM ports (jump tables and buffers) are set up, for a total of 5 additional COMs ports.
f) LPT0 jump table is loaded from $FE69. No buffer space is set up.
g) Pairs of addresses from $FF3C onwards are scanned for response, and up to two more LPT ports are defined.
h) The File Allocation Table region start is defined as the end of the LPT region. The size of this region depends on the number of drives defined in $8AFC by 'CONFIG'.
i) The FCB addresses are put into the FCB table ($8B01-$8B22).
j) The FIELD buffer address is stored in $8D40.
k) The start of user RAM can now be defined as the end of the FIELD buffer.
l) The EXEC and USR(n) jump addresses ($148-$15E) are set to 'illegal function call'.
m) The 'syntax error' table is set up from $15E-$1BD, see section 2.2.
n) The TAB table is set up, $01BE-$01C7.
o) Locations $FF38-$FF3A are scanned for a hardware clock chip and the R-C clock flag ($214) is set to $FF if the clock

is present.

p) The DOS commands table is loaded from $8A94 to $12A.
q) The jumps for the DOS extensions are set up,see section 2.2.
r) The default device location ($11B) is set to drive 0.
s) The code looks at the double byte at $8000. If it = $0C07, the initialization jumps to $8002, possibly for another type of device,or language other than BASIC??
t) Ditto for $9800, control branching to $9802. The DD DOS initialization does not branch.
u) The screen is cleared, and the keyboard, light pen and timer are enabled. Locations $2B,$3D8 are set to the top of user RAM ($8AFA).
v) Location $25 and the stack are set to 300 bytes less than $8AFA, reserving space for strings.
w) Address $8F83 is set to warm restart vector ($FE), and a NEW is performed.
x) If the program start address ($1D) is greater than $8600, then the message 'not enough memory' is displayed and the DOS hangs.
y) The NEWON flag ($248) is read and a branch to $EC49 is made if TERM mode is set.
z) Otherwise,the DISC BASIC sign-on message is displayed, and a jump to $FD42 is made to display 'bytes free' etc.

2.1.3 The sign-on message used by DISC BASIC is stored in $8A9E-$8AFB. This message can be changed to suit one's own purpose, especially if the DOS has been modified and is a non-standard version. The first byte of the message contains the number of bytes in the text.

2.1.4 Some suggestions on how to vary the startup procedure will be found in section 9.

2.2 DOS JUMPS IN COMMUNICATIONS AREA

2.2.1 There are a number of locations in the low part of RAM which ROM BASIC refers to during its execution. In a cassette system, these locations contain JMP's back to the ROM to continue cassette BASIC processing. Refer to "PEACH ROM NOTES" for details. One of the functions of the DOS initialization code is to loadthese locations with JMP's to the DOS routines which perform the DOS extended commands. A list of DD DOS JMP'S follows.

location	command	called from	JMP to (or RTS)
161	FILES	EBA6	908F
164	OPEN	E689	9312
167	CLOSE	E5FD	9577
16A	OUTPUT to file	E820	951A
16D	INPUT from file	E804	9471
170	POS function	E857	9AF8
173	EOF/LOF	E8A9	970D
176	INPUT #	D0FD	9A52
179	'Device unavailable'	--	E6AF
17C	WHILE-syntax error	--	A80F
17F	WEND- syntax error	--	A80F

182	COMMAND LINE INPUT	F4DF	8F9A
185	END OF BASIC STATEMENT	D3BA	RTS
188	DOS BASIC FUNCTIONS	D449	RTS
18B	MON	DCE6	RTS
18E	ERROR MESSAGES	A3A9	RTS
191	DISC ERROR TRAP	A3AC	8DF9
194	??	A787	RTS
197	??	D26A	RTS
19A	ROM FUNCTION ADDRESS TABLE	D39F	RTS
19D	??	D1D6	RTS
1A0	ROM FUNCTION HANDLER	D360	RTS
1A3	INITIALIZING VARIABLES	A516	RTS
1A6	TERM COMMAND	EC49	RTS
1A9	PATCH FROM A739	A739	8F62
1AC	STRING HANDLING	DE50	RTS
1AF	CTRL/C, CTRL/D	ED94	RTS
1B2	CTRL/S	F814	RTS
1B5	TYPE CHECKING	AAA2	RTS
1B8	STRING HANDLING	AD98	RTS
1BB	FILE NAME HANDLING	E405	9789

2.3 DOS MAP

ADDRESS	FUNCTION
8800-8AFB	Initialization code
8AFC-8B35	DOS variables
8B36-8C35	Primary DOS buffer (256 bytes)
8C36-8D35	Secondary DOS buffer (256 bytes)
8D36-8D4B	More DOS variables
8D4C-8D82	DOS instruction mnemonics- DSKINI,DSKI$,DSKO$,KILL,NAME,FIELD,LSET RSET,PUT,GET,VERIFY,DEVICE
8D83-8D9A	DOS instruction address table The order of the addresses corresponds to the mnemonic order
8D9B-8DB6	DOS functions mnemonics- DSKF,CVI,CVS,CVD,MKI$,MKS$,MKD$,LOC
8DB7-8DC6	DOS function address table The order of the addresses corresponds to the mnemonic order
8DC7-8DF8	Code to handle the DOS instructions and functions
8DF9-8F61	Error handling
8F62-9F6B	Code for the DOS instructions and functions
9F6A-9FF7	Not used by DOS. The directory DATE mod goes here
9FF8-9FFF	Pointers to Number of FCB's,read/write mode flag and the address of the sector read/write routine

2.4 DOS VARIABLES

2.4.1 The DOS stores a number of system related values in memory. Some relate to the system configuration while the others store more transient data. There are two main variable storage areas, one between $8AFC - $8B35, and the other between $8D36 - $8D4B. The address of these variables and their deduced meanings are listed below.

2.4.2 First variable area

ADDRESS	PURPOSE
8AFC	Number of drives - 1
8AFD	Number of FCB's - 1
8AFE	Number of 256-byte field buffers
8AFF	Size of field buffer if < 256
8B00	21. meaning unknown
8B01	Points to start of FAT region
8B03-8B22	Address of FCB's (max of 16)
8B23	I/O mode : 2=input , 3=output
8B24	Current drive number
8B25	Current track number
8B26	current sector number
8B27	Current I/O buffer address
8B29	Density flag : 0=SD , FF=DD
8B2A	Disc status register value
8B2B-8B2E	Last track position for each drive FF=off line
8B2F	Sector of directory where file name found =0 if file not found
8B30-8B31	Address in buffer of file name
8B32	current FAT value
8B33	Sector of end of directory or sector of first deleted entry
8B34-8B35	Address in buffer of end of directory or address of first deleted entry

2.4.3 Second variable area

ADDRESS	PURPOSE
8D36	Verify flag : 0=OFF , $FF=ON
8D37	Number of tries on error : =5
8D38	Number of FCB's remaining - 1
8D39	Interrupt flag : =0 if no int pending
8D3A	Address to vector to after NMI interrupt
8D3C	Copy of current DRIVE REGISTER
8D3D-8D3F	File extension
8D40	Start of field buffer,used for record I/O
8D42	Start of current record in field buffer
8D44	End of field buffer
8D46	FCB address
8D48	0=GET : 5F=PUT ,used for record I/O
8D49	Length of current record
8D4B	Density flag :- 0=SD, FF=DD

2.5 DOS USEAGE OF COMMUNICATIONS AREA RAM

2.5.1 As well as the DOS code in $8AFC - $9FF7, and the JMP table for the DOS (starting at $109), the DOS also reserves areas immediately above the screen memory for its use. These areas are used to hold the File Allocation Tables (FAT) for each disc drive defined, File Control Blocks(FCB) used for file transfer to/from the discs and the FIELD buffer used for 'record' I/O.

2.5.2 The number of FAT's and FCB's set up at initialization time depends on the configuration that has been defined for the disc currently in drive 0. The configuration for the disc is specified by using the utility program "CONFIG" supplied with the DOS. Because these reserved areas occupy valuable RAM space, the user is given the opportunity to tailor the DOS on the disc so that only the minimum of RAM space is used.

2.5.3 The number of FAT's required depends on the number of disc drives the DOS will be allowed to recognize. The minimum number of FAT's required is one, for the DOS disc itself. It is not necessary to include drive 1, but file transfer to drive 1 will then cause 'DRIVE NOT AVAILABLE' to be displayed. The number of drives - 1 is stored in location $8AFC. Each FAT occupies 162 bytes. See Section 3.2 for further discussion on FAT's.

2.5.4 Each open file must have an FCB associated with it, to hold data being read from or written to the file. The maximum number of files that may be open at any given time is 16. The default number is 4 but "CONFIG" allows this number to be altered. The number of FCB's - 1 is stored in $8AFD. Each FCB occupies 281 bytes i.e. a 25 byte header and a 256 byte buffer. Section 3.3 discusses FCB's further.

2.5.5 "CONFIG" also allows the size of the FIELD buffer to be defined. It is usual to have a buffer size of 128 or 256 bytes, but up to 2048 bytes may be reserved. The address of the start of the FIELD buffer is stored in $8D40 and the end in $8D44. $8D42 points to the start of the currently referred to record in the FIELD buffer.

2.5.6 The DOS useage of RAM above the screen memory thus consists of the following reserved areas:-

```
USER RAM
FIELD buffer
FCB n
FCB n-1
.
.
FCB 1
FAT Drive number n
.
.
FAT Drive number 1
FAT Drive number 0
screen RAM
```

3.0 DISC FILE SYSTEM

3.0.1 Files are stored on disc in multiples of groups, where each group is 8 sectors . the names of the files and some information about the file is stored in a directory. The groups used by each file are recorded in a File Allocation Table. When files are accessed, the data transferred is buffered in a File Control Block. Record mode files use a FIELD buffer to construct records.

3.1 DISC DIRECTORY

3.1.1 The disc directory resides on track 20 of each disc, starting from sector 5. The directory is initialized by a DSKINI instruction, section 7.1. Each directory entry occupies 32 bytes so that 8 file names are stored on each sector. The directory extends to sector 32 but a maximum of 152 files only is allowed. The directory effectively ends at sector 24.

3.1.2 The end of directory is indicated by an $FF byte at the beginning of the next file name entry. A blank directory contains all $FF's.

3.1.3 When a file name is deleted from the directory, a 00 byte is put in the first character position of the file name. When the DOS looks for space in the directory to put a new file, it finds either the end of directory, or the first deleted file name it comes to. New files thus do not always appear at the end of the directory listing.

3.1.4 The directory listing is displayed by the FILES command. This is not a new DOS command but an extension to the cassette system command. The code for the FILES command begins at $908F and is called from $EBA6 via $161.

3.1.5 The structure of each directory entry is as follows:-

BYTES	MEANING
0 - 7	File name
8 - 10	Extension
11	Kind, 0=BAS, 1=DAT, 2=BIN
12	Type, $0=Tokenized BASIC, $FF=ASCII
13	FAT address of first group of the file
14	unused
15	Number of bytes used in last sector
16-31	unused

3.1.6 As not all the space in each file name entry in the directory is used, it is possible to store there additional information about the file. Refer to section 9 for more details.

3.2 FILE ALLOCATION TABLE (FAT)

3.2.1 The DOS maintains a table on track 20,sector 2 of each disc containing information about where each sector of each file is stored. However, rather than keep a complete map of every sector on the disc(1280), the storage requirement is reduced by defining the

smallest storage segment as 8 sectors. This is called a group.Thus, with 40 tracks and 32 sectors per track, the maximum number of groups on a disc is 160. The first 16 sectors of track 0 are single density, with the DOS boot loader program on sector 1. Although sectors 17-32 are double density, the whole of track 0 is not available for file storage. Track 20 is also not available for file storage as this holds the FAT and file directory. If no DOS is on the disc, 152 groups are free. However, if a DOS resides on the disc, sectors 1-24 of track 1 are used to hold the DOS code, reducing the number of groups available by 3, to 149. The FAT is set up to handle 152 groups.

3.2.2 A free or un-allocated group is indicated by an $FF value in the FAT. The DOS track is indicated by an $FE value. The directory track (20) is not represented in the FAT but is allowed for by adding 1 to the computed track if it is greater than 19. A FAT value less than $C0 indicates the group is part of a file,and points to the next group of the file. A FAT value of $C0 - $C8 means that the group is the last one in the file. The second digit is the number of sectors used in the group.

3.2.3 The current FAT value (if < $C0), points to the address in the FAT where the next FAT value may be found.

3.2.4 The 15th byte of each file name record in the directory points to the address in the FAT of the first group of the file. This can be decoded into the track and sector number of the first sector of the file, as follows. As there are 4 groups on a track, the FAT address is divided by 4 to obtain the track number. To protect the directory, as already explained, the track number is incremented by 1 if >= 20.

3.2.5 The sector number is computed from the remainder after the FAT address is divided by 4. This remainder (0-3) determines the group number in the track. The remainder is thus multiplied by 8 and then incremented by 1 to produce the sector number.

3.2.6 The FAT for each disc is stored in RAM immediately above the screen RAM. Each FAT occupies $9E (158) locations,i.e 152 group values plus a 6 byte header. The use of this header is unclear,except that the first location keeps a count of the number of files open for the disc.

3.2.7 Because the directory is in the middle of the disc (track 20), it makes sense to store files close by to reduce seek times. The DOS therefore allocates groups, starting with track 19, then track 21, then 18, then 22 etc, spreading out from track 20. As the utility routines are usually the first files to be put on a disc, tracks 18,19 and 21 are referred to in the DOS manual as the utility tracks.

3.3 FILE CONTROL BLOCKS (FCB)

3.3.1 When ever a file is opened for access, an FCB is created. This FCB consists of a 25 byte header and a 256 byte buffer. There is a limit to how many FCB's may exist at any one time, since they take up RAM space which could be used for programs. This limit is set by

the utility program "CONFIG" when it asks how many files are allowed to be open at any time.

3.3.2 The addresses of the FCB's are held in a list in the DOS, at $8B03-$8B22. The FCB addresses depend on the screen resolution, since the FCB area is above the screen RAM.

3.3.3 The FCB's are allocated from the last to the first, and follows from the fact that the number stored in the open-files table is the number of FCB's remaining.

3.3.4 The meanings of the variables stored in the header have not all been decyphered, but some are known quite definitely. The meanings differ depending on the mode of the data transfer i.e INPUT,OUTPUT, RECORD or DATA modes.

3.3.5 Relative byte 12 contains a hash value which is derived from the sector number of the directory sector containing the file name and it's address within the sector, according to the following formula:-

Hash=(sect-5)*8 + (buffer start-entry address)/32

This hash value is used by the CLOSE command to enter the number of bytes used in the last sector into the directory. This method must be used since the file name is not kept in the FCB, so the normal directory search for the file name entry can not be used.

3.3.6 INPUT MODE FCB

OFFSET(hex)	MEANING
00	Mode, = $10 for INPUT
01	Drive number
02	FAT value in directory
03	Current FAT value
04	Relative sector in current group
05	Number of bytes read out so far
06	unused
07	Relative sector of next sector
08	unused
09	unused
0A	unused
0B	unused
0C	unused
0D	unused
0E	unused
0F	unused
10	Last char flag????
11	Last char????
12	Directory sector/address hash value
13	Number of bytes in last sector
14	????
15	unused
16	unused
17	EOF flag, = $FF if EOF, else = 0
18	Number of chars left in buffer

19	Start of data in buffer
.	.
.	.
118	End of buffer

3.3.7 OUTPUT MODE FCB

OFFSET(hex)	MEANING
00	Mode, = $20 for output
01	Drive number
02	FAT value in directory
03	Current FAT value
04	Relative sector in current group
05	unused
06	Number of chars put in buffer since last CR - chars < $20 are not counted
07	Relative sector of next sector
08	unused
09	unused
0A	unused
0B	unused
0C	unused
0D	unused
0E	unused
0F	unused
10	unused
11	Terminating char ????
12	Directory sector/address hash value
13	Number of bytes in last sector
14	unused
15	unused
16	unused
17	unused
18	Number of chars in buffer
19	Start of data in buffer
.	.
.	.
118	End of buffer

3.3.8 RECORD OR DATA MODE FCB

OFFSET(hex)	MEANING
00	Mode, = $40 for RECORD or DATA mode
01	Drive number
02	FAT value in directory
03	Current FAT value
04	Relative sector in current group
05	unused
06	?????
07	Record number for next access
08	
09	Length of record
0A	
0B	Start of record in field buffer

0C	
0D	?????
0E	unused
0F	unused
10	Get/Put flag, Get=0, Put=$5F
11	unused
12	unused
13	unused
14	unused
15	cleared but unused
16	cleared but unused
17	cleared but unused
18	cleared but unused
19	Start of data in buffer
.	.
.	.
118	End of buffer

4.0 READING FROM AND WRITING TO DISC

4.0.1 When using the disc for input/output, it is important to preserve the integrity of the directory system. This means that any writing to the disc should always be done by DOS routines, or at least by routines that obey the directory rules. In the discussion that follows, this principle will be followed.

4.0.2 In spite of what has just been said, the DOS supports direct reading and writing from/to sectors. The user could write programs which do not use the directory, but it would be sensible to use discs reserved for this direct access method. This type of access would be useful for reading or writing discs in a non-PEACH format.

4.0.3 Disc I/O is an extension of the cassette I/O system implemented in the ROM. Each major command such as OPEN, has a JMP to a location in the communications area which, in the cassette system, either causes a SYNTAX ERROR, or returns to further process the cassette command. When the DOS initializes itself, these JMP locations are set up to point to the DOS routines which will execute the DOS extensions.

4.0.4 Disc I/O can be implemented by directly calling the ROM subroutines which perform the OPEN,CLOSE,OUTPUT,INPUT commands (as is detailed in PEACH ROM NOTES). A file name is required for the OPEN command, but once the file is opened, it is referred to by its file number, stored in $9E. Each open file has a corresponding FCB which is located by accessing the OPEN-FILES table by file number.

4.1 OPEN FILE COMMAND

4.1.1 This command is processed by the ROM BASIC code until the file description is required. A JMP to $1BB is executed which directs the processing to the DOS code at $9789, where the disc drive, file name, and extension are extracted from the command. Having obtained the file descriptor, the DOS code at $9312 is then executed via $164, to open the file.

4.1.2 The OPEN command has three varients- open for INPUT, open for OUTPUT, or open for RECORDs or DATA. The opening process consists of setting up a File Control Block for the file, up to the limit of the maximum number of open files allowed (see section 2.5.2). The OPEN-FILES table is set up, equating the file number with the FCB number.

4.1.3 If the file is opened for INPUT, the directory is searched for the file name. If found, the first sector of the file is read into the FCB buffer ready for subsequent input requests.

4.1.4 If the file is opened for OUTPUT, the directory is scanned for the file name. If found, the file is killed and a new one opened.

4.1.5 RECORD or DATA mode does not delete a file if it already exists. If the file exists and no FCB is already opened, a new FCB is opened.

4.1.6 A useful technique for opening a file from a machine code program is to get BASIC to perform the OPEN command, followed by a return to the calling program. The following code is an example of how this may be done.

```
        LDX   #CMD          POINT TO COMMAND
        LDU   #$278         POINT TO BASIC'S COMMAND LINE BUFFER
        LDB   #$21          NUMBER OF BYTES IN CMD(INCL 00)
        JSR   $FC48         MOVE INTO BUFFER
        LDX   #$278
        STX   $BC
        STS   STACK         STACK IS NOT PRESERVED BY BASIC
        CLRB
        TFR   B,DP
        JMP   $A450         JMP TO BASIC
*
*NOTE:: NNNN MUST CORRESPOND TO THE RETURN ADDRESS, 'NEXT'
*
CMD     FCC   'OPEN"O",8,"NAME":EXEC&HNNNN'
        FCB   00
STACK   FDB   0000          SAVE STACK HERE
*
*RETURN HERE AFTER BASIC
*
NEXT    LDS   STACK         RESTORE STACK
        .....
```

4.2 CLOSE FILE COMMAND

4.2.1 The code for the CLOSE command starts at $9577 and is reached via $167 from $E5FD. If the file to be closed is an INPUT file, the closing process involves recovering the used FCB and writing out the FAT. The first location of the FCB header, the I/O mode, is set to 0.

4.2.2 If the file has been used for OUTPUT, the remaining data in the FCB buffer is written to the disc and the directory is updated to record the number of bytes occupied on the last sector. The FCB space is recovered and the FAT written to the disc.

4.2.3 If in RECORD or DATA mode, the closing process is more complicated, as fielded strings have to be moved to the FCB buffer before the last record is written. Also, strings that have their address pointers pointing to the record area in the FIELD buffer are set to NULL strings. The FAT and directory are then updated on disc.

4.2.4 Files can be closed from a machine code program by setting up the required BASIC command as explained above for OPEN. Alternatively, there is a CLOSE-ALL-FILES routine at $E62B which is simply called as a subroutine, and is easier to use if all open files can be closed at the same time.

4.3 INPUT FROM FILE

4.3.1 THis code at $9471 is reached from the generalized input routine at $E804 via $16D. The B register contains the FCB number. A character (in the A register) is handed to the calling routine every time the input routine is called.

4.3.2 The FCB is accessed for the next character. If EOF has been reached, location $9F is set to $FF. If the FCB buffer has only one character left, this character is held while the next sector is read in. The held character is returned as the next character. If there is more than one character left, the next character is read from the FCB buffer and the pointer to the next character is moved on one.

4.3.3 A typical code sequence for inputting a character from the disc is as follows, assuming the file has already been opened for INPUT. The character read from the file will be returned in the A register.

```
        CLRB
        TFR   B,DP          ONLY IF DP NOT = 0
        LDB   #$...         FILE NUMBER
        JSR   $EB04         GENERALIZED INPUT ROUTINE
        TST   $9F           EOF?
        BNE   ...           YES,PROCESS EOF
        ...                 NO,PROCESS CHAR
        ...
```

4.4 OUTPUT TO FILE

4.4.1 This code is reached from the generalized output routine at $E820 via $16A. The B register contains the FCB number. A character is accepted in the A register and is entered into the FCB buffer.

4.4.2 When the FCB is filled by the new character, the buffer is written to the disc. The FCB keeps a count of the number of characters since the last CR. Control characters (< space in magnitude) are not counted, but they are put into the buffer.

4.4.3 The DOS variable at $8D36 can be set to $FF if verification is desired when writing to the disc. $8D36 is initialized to $FF (on) on boot up. Sector writes are slower if verify is on.

4.4.4 A typical way to output characters to disc, is as follows, assuming the file has been opened for OUTPUT. The character to be output must be in the A register.

```
        CLRB
        TFR   B,DP          ONLY IF DP NOT = 0
        LDB   #$..          FILE NUMBER
        JSR   $E820         GENERALIZED OUTPUT ROUTINE
        TST   $9F           GOOD WRITE?
        BNE   ...           NO,PROCESS BAD WRITE
        ....                GET NEXT CHARACTER
```

4.5 RECORD MODE FILE ACCESS

4.5.1 The above input and output processes apply to sequential files, i.e. ones that have to be read sequentially from beginning to end. No information is known about where data is stored in the file, so the file must be read sector by sector until the required data is found. Data cannot be inserted directly into these files. To add data, a new file must be created containing data up to the point where new data us to be added, the new data can then be output to the new file, then the rest of the old file can be output to the new one. Record mode files avoid this problem by allowing reading and writing directly to numbered records, no matter where they are in the file.

4.5.2 For each record mode file created, a fixed length record is defined. Each record to be output is constructed in the FIELD buffer using the form defined by a FIELD statement in BASIC, and with length defined in the OPEN command.

4.5.3 A number of files may be open for record I/O at any one time, but there is only one FIELD buffer. Record structures are kept distinct by storing the start of a file's record area, and the record length, in the FCB header. The start of the current record is stored in $8D42, and the length in $8D49.

4.5.4 The sum of the lengths of all the records currently open must be less than the length of the FIELD buffer. For example,

```
End of FIELD buffer
      .
      :
End of record for #2
      .
End of record for #1,start of record for #2
      .
Start of FIELD buffer, start of record for #1
```

4.5.5 Record lengths do not have to be restricted to multiples of powers of 2, but can be any length up to the length of the FIELD buffer. The FIELD buffer length is defined by "CONFIG" and may be up to 2048 bytes long. This means that records can straddle sector boundaries.

4.5.6 Records are not stored in separate sectors, but are packed into the sectors without any wasted space between adjacent records. The records themselves may have unused space in them, but record boundaries have no gaps between them.

4.5.7 When accessing a record, the DOS has to calculate the sector that the required record is on, and where in the sector the record starts.

4.5.8 If the next record to be output has a record number greater than the biggest record number yet written, the DOS fills the sectors in between with '@' characters. It is possible to reserve disc space for the whole file by PUT-ing to the disc a record with the maximum record number which will be used.

4.5.9 When a string appears in a FIELD statement, it is placed in BASIC's symbol table. The length of the string is taken from the FIELD statement, with the address of the string pointing to the position in the record in the FIELD buffer where the string will be stored. Until an LSET or RSET statement is executed, the strings will have nothing in them. Only when an LSET or RSET command is processed will the appropiate string contents be moved into the relevent position in the record in the FIELD buffer.

4.5.10 After a record is output with a PUT statement, the symbol table is searched for any strings or string array elements that point into the record area, and they are set to NULL strings i.e. any string that appeared in the appropriate FIELD statement.

4.5.11 It is apparent that if a file is opened with a record structure that is not the same as that used in constructing the file, nonsense will result.

4.5.12 It is possible, however, to have two FIELD statements referring to the one record, with the second FIELD statement referring to sub-fields of the first record. This is sometimes convenient to use, as the following example shows.

```
100 FIELD #1,100 AS A$
105 GET #1,RECNO
110 FIELD #1,20 AS NAM$,40 AS AD$,20 AS NAT$,10 AS SEX$,10 AS TEL$
115 PRINT NAM$: PRINT AD$: PRINT NAT$: PRINT SEX$: PRINT TEL$
```

5.0 FILE STRUCTURES

5.0.1 When sequential files are stored on the disc under BASIC, the first byte of the first sector defines the type of file. A $00 indicates a binary file, while BASIC files can start with $FF, or $0D.

5.1 BINARY FILES

5.1.1 There are 4 other bytes at the beginning of the binary file, and an additional 5 bytes at the end of the file, that are not data bytes, but control bytes for the LOADM command's use.

5.1.2 The second and third bytes contain the count of the number of data bytes in the file. This number does not include any of the 10 control bytes. The fourth and fifth bytes define the first address that the file is to be loaded into.

5.1.3 The 2-byte entry point address of the binary file is stored at the end of the data on the last sector, and is preceded by $FF, $00, $00. The entry point address is placed in location $148 by the LOADM command and is the address jumped to when the R option is used with that command.

5.1.4 The binary file thus has the following structure.

BINARY PROG LAYOUT

BYTE SEQUENCE	MEANING
00	binary file indicator
nnnn	2-byte data count
NNNN	2-byte address where the file is loaded
.	binary data
.	
.	
.	last data byte
FF	entry point header
00	
00	
XXXX	entry point address

5.2 BASIC FILES

5.2.1 The first byte of a BASIC file contains either $FF or $0D. $FF means that the file is in tokenized form, whereas $0D indicates it is in ASCII form. The ASCII form is equivalent to a LIST.

5.2.2 The next 2 bytes of a tokenized file represent a line number, above which BASIC will not display with the LIST command. This line number is normally $FFFF, so all lines will be listed. However, by use of the UNLIST command before the program is SAVEd, some measure of protection can be given from prying eyes.

5.2.3 A tokenized file is stored with all the address pointers to subsequent lines, exactly as the program exists in RAM. However, if it is LOADed back into RAM at some future time with a NEWON value that is different from that when it was SAVEd, the addresses will be wrong, as the program will be loaded into a different part of RAM. BASIC

accounts for this and inserts the correct address links for the new RAM location.

5.3 RECORD FILES

5.3.1 Record files have no control bytes at the start of the file. The first record thus begins at byte 0 of the first sector. Subsequent records begin at locations that are multiples of the record length used to set up the file. Record 2, therefore, would begin in the first sector if the record length was less than 256. If the record length was greater than 256, the next record would begin in a subsequent sector.

5.3.2 Records that have not had any data output to them, and which lie between already defined records, are filled with $40's. Records beyond the one with the largest record number so far defined, do not exist in the file structure. Space for all the records to be used in the file can be reserved by outPUT-ing a record with the largest record number to be used, to the file.

6.0 DOS SUBROUTINES

6.0.1 There are a number of DOS subroutines which may be useful to the assembly langauge programmer. In the discussion that follows, the variables which must be set up before using the routines, and those whose values are altered by the routine, are listed.

6.1 'ARE YOU SURE?' MESSAGE - $904F

6.1.1 This routine may be useful when requesting confirmation by the user of some requested action. The message 'Are you sure (Y or N)?' is displayed and the keyboard is read for a reply. If 'Y', the EQUAL flag is set. If 'N', it is cleared. If any other character is entered, the message is repeated.

6.2 COUNT THE NUMBER OF GROUPS IN FILE - $9102

B reg	FAT address of first group of file
$8B24	Drive number
A reg	The number of groups in the file

6.2.1 The FILES command uses this routine. The address of the appropriate FAT in RAM is found and the FAT scanned from group to group until the end group is reached. The X register will point to this last group in the FAT, and the B register will have the last group value.

6.2.2 This routine may be useful if the address of the last group of a file is wanted, e.g. when determining the entry address of a binary file. (This address is saved as the last two bytes of the last sector,section 5.1).

6.3 COUNT THE NUMBER OF GROUPS LEFT ON DISC - $9102

6.3.1 This routine returns to BASIC after its task is completed. However it is a very useful routine so the code is repeated here as a subroutine that returns the group count in the B register.

$8B24	Drive number
B reg	Count returned

```
        *
        *count # groups free on disc
        *return result in B reg
        *
GPSFR   JSR   $9A02     READ IN FAT FROM DISC
        JSR   $98C4     POINT TO FAT (X REG)
        LEAX  $06,X     POINT PAST HEADER
        CLR   ,-S       STORE COUNT ON STACK
        LDB   #$98      MAX # GROUPS
GPS1    LDA   ,X+       GET fat VALUE
        COMA            IS IT $FF ?
        BNE   GPS2      NO,NOT FREE
        INC   ,S        YES,COUNT IT
```



```
GPS2    DECB
        BNE   GPS1      LOOP FOR ALL GROUPS
        PULS  B
        RTS
```

6.4 FIND FILE NAME IN DIRECTORY - $98D3

$01FE	File name (max 8 chars,padded with spaces)
$8D3D	File extension (max 3 chars,padded with spaces)
$8B2F	Directory sector where file name found
$8B30	Address in primary buffer where file name found
$8B32	FAT value of first group of file
$8B33	sector of end of directory,or first deleted entry
$8B34	Address in primary buffer of end,or first deleted entry

6.4.1 The routine reads each directory sector into the primary buffer until the file name and extension matches with an entry in the directory, or the end of directory is reached. If a match is found, $8B2F, $8B30 are updated. If no match is found, $8B2F will be 0.

6.4.2 The routine also stores the sector and address within the primary buffer of the first deleted entry it comes to, or the end of the directory.

6.5 FIND FREE SPACE IN FAT - $9948

B reg	FAT value to start scanning from
$8B24	Disc drive number

6.5.1 This routine reads in the FAT from the drive defined in $8B24, and scans it for a free space i.e. for an $FF. The scaning begins at a point in the FAT defined by the B register. When a free space is found the value $C0 is stored in the same location in the FAT in RAM. The X register will point to this location on exit.

6.5.2 If no space is available the error message DISC FULL is displayed and control returns to BASIC.

6.6 READ FAT FROM DISC - $99EF

$206	Device number
$8B24	Drive number

6.6.1 The device number in $206 is checked for the most significant bit set, to indicate disc I/O. The drive number is extracted from $206 and stored in $8B24. The FAT is then read from the appropriate drive and moved into the FAT storage area in RAM.

6.7 READ/WRITE FROM/TO DISC SECTOR WITH VERIFY - $9D6B

$8B23	Read/write mode:- read=2, write=3
$8B24	Drive number
$8B25	Track number
$8B26	Sector number

```
$8B27     Buffer address to read into or write from
$8B29     DD flag:- DD=$FF, SD=0
$8B2A     Disc status value
$8D36     Verify flag:- verify on=$FF, off=$00
```

6.7.1 This routine will read or write a sector,according to the setting of the above variables.

6.7.2 If in write mode and verify is on, the routine will read back, into the secondary buffer ($8C36-$8D35), the sector just written and compare the data read back, with the data held in the input buffer. It will re-try five times until no error occurs. A VERIFICATION FAILURE will be displayed if still in error, and control returns to BASIC. Writes to disc will obviously be slightly slower if verify is on.

6.7.3 If the status variable $8B2A is not equal to 0 on return from the main read/write routine($9DE7) one of the error messages "DRIVE NOT READY", "DISC WRITE PROTECTED",or "DEVICE I/O ERROR" will be displayed and control will return to BASIC.

6.8 MAIN DISC READ/WRITE ROUTINE - $9DE7

6.8.1 This is the routine that does all the work, according to the variables (except verify) already defined in section 6.7. No error decoding is made.

6.8.2 The code expects the disc registers to be $FF00-$FF04. Because the DATA BUS is inverted between the CPU and the drives, the data sent to or read from the disc registers must be inverted. The meaning of these registers is as follows.

address	R/W	MEANING
$FF00	read	status (see section 6.8.10)
	write	command to controller
$FF01	read/write	current track
$FF02	read/write	required sector
$FF03	read/write	data to/from disc
$FF04	read	auxilliary status: bit7=NOT(DRQ) bit1-6=unused bit0=NOT(IRQ)
	write	mode select: bit7=unused bit6=NMI mask bit5=SD/DD DD=1,SD=0 bit4=SIDE: 0=0,1=1 bit3=MOTOR ON: on=1 bit2=unused bit0,1=DRIVE SELECT

6.8.3 The HITACHI disc controller IC is equivalent to the WESTERN DIGITAL 1791 floppy disc controller. The commands used by the DOS I/O routine are detailed in the following table.

```
01        restore,12 ms stepping rate
11        seek,12 ms
80        read,single sector,no 15 ms delay
A0        write,single sector,no 15 ms delay
D0        clear,no interrupt
```


6.8.4 If the drive is not ready, the routine times out after approximately 20 seconds. This gives time to insert a disc if one is not already in.

6.8.5 After the drive is started and up to speed, the code selects from an address table, one of four I/O routines to execute. The index for this selection is the I/O mode variable $8B23. Note, however, that the DOS only ever uses two of these possible four modes, with 2=READ, 3=WRITE. Mode=0 causes the selected drive to restore the head to track 0, while mode=1 executes an RTS and does nothing.

6.8.6 DD disc I/O uses the NMI interrupt for finding the required track and sector, as does the SD DOS. However, data transfer to/from the disc is performed character by character under program control, by making use of the DATA REQUEST (DRQ) bit in $FF04. This bit is set by the disc controller whenever it needs another character (write), or when it has another character available (read). The program simply sits in a loop continually examining this bit, and supplying the next character, or storing the next character when necessary.

6.8.7 In SD DOS, the characters are counted and the loop ends when 128 are transferred. However, in DD DOS, no counting is done. Use is made of the fact that the controller will raise an NMI interrupt when the end of sector is reached. Hence, before the sector transfer begins, the address to vector to when the interrupt is received, is set up in $1A9. This address points to the code to execute to check that a good transfer has been made.

6.8.8 This technique has the advantage that the routine can be used for reading sectors of any length. The only difference between single or double density I/O is bit 5 of $FF04. The controller sorts out the disc format. It seems possible to have discs formatted in (say) 512 bytes per sector, and have this routine read them.

6.8.9 The DD DOS disc routine turns off the IRQ and FIRQ interrupts while data is being transferred. This prevents the keyboard from interfering with the I/O process, as is the case with the SD DOS.

6.8.10 Error conditions are returned in $8B2A and have the following meaning.

BIT	ERROR MEANING
7	not ready
6	write protect
5	-
4	not found or seek error
3	CRC error
2	lost data
1	-
0	-

7.0 NOTES ON DOS COMMANDS

7.0.1 Apart from extensions to the cassette system commands, the DOS provides additional commands for specific disc actions. These commands are referenced from a table of pointers loaded into RAM at the end of the BASIC command pointers. As explained in PEACH ROM NOTES, the ROM command processing code at $D40D, is set up to look for further commands if not found in the first table. The DOS pointer table loads into $12A from $8A94 during initialization and has the following values.

$12A	$0C	number of disc instructions
$12B	$8D4C	address of instruction names list
$12D	$8D37	instruction processing routine address
$12F	$08	number of disc functions
$130	$8D9B	address of function names list
$132	$8DDE	function processing routine address

7.0.2 The DOS instruction processing routine at $8D37, checks for a token value between $DD and $E8. If greater than $E8, the code jumps to $137 in case another instruction pointer table exists. For standard DOS, $137 points to the 'SYNTAX ERROR'message.

7.0.3 The function processing routine simply checks for a token between $54 and $62. If greater than $62, a jump to $13C is made to look for a further table.

7.0.4 It should be noted that these processing routines only check that the token value is legal. The ROM code at $D40D then picks the appropriate subroutine address to execute the command, from the address list pointed to by the command address table. When tokenizing a BASIC statement, the ROM refers directly to the command table.

7.0.5 The following information on each of the DOS commands may be useful in understanding how the command operates. This section does not replace the DOS Manual, and assumes the user had read the manual.

7.0.6 The address listed with each command is the starting address in RAM of the command.

7.1 DSKINI - $8FBB

7.1.1 This code initializes a formatted disc by writing a blank directory and FAT on track 20, sectors 1 to 32. All bytes are set to $FF. The process takes longer than the SD command, as there are twice as many tracks and they contain twice as many bytes.

7.2 DSKI$ - $920C

7.2.1 This INSTRUCTION is a FUNCTION in Single Density DOS. If only one string is defined in the BASIC statement, the sector is read in single density,i.e. a string of 128 bytes.

7.2.2 The required sector is read into the DOS primary buffer($8B36-$8C35) and then moved into BASIC's string space. The string pointer in BASIC's symbol table is set to point to the string. As this operation is directly applied to a specific sector, no FCB is involved,i.e it is not necessary to open a file to use this instruction.

7.3 DSKO$ - $92A6

7.3.1 The same comments for DSKI$ also apply to DSKO$. The only difference is that DSKO$ moves one or two strings from the string space to disc.

7.4 KILL - $9119

7.4.1 This writes a $00 byte in the first character of the file name in the directory. The FAT is then searched and all groups used by the file are recovered by setting their FAT value to $FF.

7.4.2 It is possible to recover the deleted file because the FAT value in the directory is not altered by the KILL process. This FAT value points to the track and sector of the first sector of the file. If the disc has not been altered since the file was KILLED, the sectors storing the file will still be intact. By examining the disc sector by sector and rewriting the FAT, the file structure can be reconstructed. However it is not easy,and one needs a good sector read/write program to help.

7.5 NAME - $91A3

7.5.1 The name of a file may be changed by altering the file name and extension in the directory. Nothing else about the file is altered.

7.6 FIELD - $9B07

7.6.1 This instruction operates by inserting in BASIC's symbol table, strings (whose names are defined in the FIELD command) with pointers to the start of each record in the FIELD buffer. Refer to Section 4.5 for more discussion.

7.7 RSET,LSET - $9B45,$9B46

7.7.1 These commands move the second string defined in the command to the first string in the command. The first string named must have a pointer within the record in the FIELD buffer that the file is using. That is, the string must have been defined by a FIELD command prior to the RSET or LSET command.

7.7.2 The region that the second string is loaded into is cleared before transfer.

7.7.3 The same code is used for both LSET and RSET, except the string when it is moved into the record is left justified in the case of the LSET and right justified in the case of RSET.

7.8 PUT,GET - $9B9F,$9BA0

7.8.1 The same code is shared by GET and PUT commands. The flag $8D48 is used to jump over irrelevant code. $8D48 = 0 for GET and $5F for PUT. See Section 4.5 for more discussion on record mode I/O.

7.8.2 The file must be open for 'record' mode so that an FCB can be opened. The current record number is stored in the FCB header. See Section 3.3 for a discussion of the FCB structure.

7.9 VERIFY - $9192

7.9.1 Verify sets a flag ($8D36) to $FF if writing to the disc is to be verified, and to $00 if no verification is required. Refer to Section 6.7 for a discussion of the verification process.

7.10 DEVICE - $917B

7.10.1 The DEVICE command allows the default I/O device to be redefined. The default device set up at boot time is drive 0. Location $11B contains this device number ($02=cassette, $80=drive 0). This command is useful if a lot of use is to be made of (say) the cassette.

7.11 DSKF function - $96B6

7.11.1 This computes the number of free groups on the disc. The FAT is scanned and the integer result is returned to BASIC by a JMP to $ACE6. The result is placed in $58,$59 by $ACE6.

7.12 CVI,CVS,CVD functions - $9D34,$9D37,$9D3A

7.12.1 These three functions share similar code. They set the B register to 2,4 or 8 respv then JSR to $AEB7 which does the conversion.

7.13 MKI$,MKS$,MKD$ functions - $9D4E,$9D51,$9D54

7.13.1 These three functions also share similar code. They set the B register to 2,4, or 8 respv then JSR to $AD78 which converts the numbers to character strings of 2,4, or 8 bytes.

7.14 LOC - $96DE

7.14.1 This code gets the next record number of an open file from its FCB and returns to BASIC with a JMP to $ACE7 which stores the D register in $58, $59.

7.15 LOF/EOF - $970D

7.15.1 This code is reached from $E8A9 via $173. If EOF is requested, the FCB is first checked for input mode, then relative byte $10 of the FCB is examined. If = 0, there are still some bytes in the buffer, and the number remaining,(held in relative byte $17) is returned to BASIC. If not = 0, the buffer is empty, and $FF is returned to BASIC as EOF.

7.15.2 LOF returns the number of groups in the file, unless in 'R' mode. In 'R' mode, the largest record number used is returned to BASIC.

8.0 DD DISC FORMAT

8.0.1 The floppy disc controller used in the PEACH is equivalent to the WESTERN DIGITAL 1791 IC which can support a number of different DD formats, as well as SD.

8.0.2 The disc is formatted in terms of concentric tracks made up of a number of sectors. Data can be read from or written to the disc only in whole sectors.

8.0.3 The HITACHI DD format consists of 40 tracks (numbered 0 to 39) on each side. There are 16 sectors (numbered 1 to 16) in each track on EACH side. The DD DOS considers tracks to have a total of 32 sectors, with sectors 1 to 16 on side 0, and sectors 17 to 32 on side 1.

8.0.4 Each sector has a header defining the track number, side number, sector number, and number of data bytes.

8.0.5 Sectors are separated by gaps containing control bytes designed to enable the controller to synchronize to the pattern. This is necessary because if the disc speed varies, the physical length of the data bytes will be slightly longer or shorter every time they are re-written. Thus there may be some left-over bits at the end of the data area in each sector. The controller therefore must resynchronize after reading every sector.

8.0.6 The number of data bytes in the sector is flagged in the sector header. This flag can have 4 values as the following table shows. The DD sectors use a value of 01, with SD sectors using 00.

Sector flag	Number of bytes in sector
00	128
01	256
02	512
03	1024

8.0.7 The sectors are not stored in consecutive order on a track. The order for the DD discs is 1,9,4,12,7,15,2,10,5,13,8,16,3,11,6,14. This may seem strange, but it will be noticed that there are 5 sectors between consecutively numbered sectors. This gives time for a program to process one sector before the next one comes under the head. You will have noticed that FORMAT uses a machine code program called SKIP5. No marks for guessing the origin of the name!

8.0.8 The disc rotates at approximately 300 rpm. One revolution therefore takes 3.33 ms. One sector is under the head for 0.208 ms. 5 sectors are scanned in 1.04 ms. This then is the maximum time a program can spend processing a sector if it is to ready for the next sector before it passes the head.

8.0.9 Discs are formatted by writing whole tracks in one go. The track data is set up in memory in one long buffer and written onto the disc by issueing a TRACK WRITE command to the disc controller.

8.0.10 All the tracks except track 0 are formatted for DD storage. Track 0 is different, in that the first 16 sectors (ie side 0) are formatted in single density, while the rest of the track (on side 1) is in DD mode.

8.0.11 The SD disc format (for sectors 1-16,track 0) is as follows:-

	Number of bytes (DEC)	Value of byte written (HEX)		
	40	FF	)	
	6	00	)-	TRACK HEADER
	1	FC Index mark	)	
	26	FF separator	)	
	(6	00	)	
	(1	FE ID mark	)	
	(1	<track number>	)-	SECTOR HEADER
	(1	<side number>	)	
	(1	<sector number>	)	
	(1	00 sector length	)	
repeat for 16-sectors	(1	F7 2 CRC's written	)	
	(12	FF separator	)	
	(6	00	)	
	(1	FB data mark	)-	SECTOR DATA
	(128	<data bytes>	)	
	(1	F7 2 CRC's written	)	
	(28	FF separator	)	
	247	FF separator	)-	END OF TRACK

8.0.12 The DD disc format for the rest of the disc is as follows:-

	Number of bytes (DEC)	Value of byte written (HEX)		
	80	4E	)	
	12	00	)	
	3	F6	)-	TRACK HEADER
	1	FC Index mark	)	
	50	4E separator	)	
	(12	00	)	
	(3	F5	)	
	(1	FE ID mark	)	
	(1	<track number>	)-	SECTOR HEADER
	(1	<side number>	)	
	(1	<sector number>	)	
	(1	01 sector length	)	
repeat for 16-sectors	(1	F7 2 CRC's written	)	
	(23	4E separator	)	

(12	00	)	
(3	F5	)	
(1	FB data mark	)-	SECTOR DATA
(256	<data bytes>	)	
(1	F7 2 CRC's written	)	
(55	4E separator	)	
598	4E separator	)-	END OF TRACK

8.0.13 Note that when the controller receives an $F7 byte it automatically generates a 2-byte CRC (Cyclic Redundancy Checksum). This CRC is recalculated by the controller when the sector is read and the CRC error bit in the status byte set accordingly. The disc routines do not have to concern themselves with this CRC generation, only if it is correct or otherwise.

8.0.14 The WRITE TRACK command writes data continually from the index hole on the disc to the next time the index hole is reached. The number of bytes specified as the end of track separators are only approximate. More than this amount should be provided for so that the index hole is always reached.

9.0 DOS WEAKNESSES AND IMPROVEMENTS

9.0.1 There are a number of areas where the DOS is lacking. While the DOS seems to be well written and bug free, a number of improvements could be made to increase the utility of the DOS, and make it more versatile.

9.0.2 There is some unused space at the end of the DOS, from $9F6A-$9FF7, which can be used to hold some code. This has already been used to good effect by some of the improvements to be discussed. However, this space is very limited, and other techniques will also be discussed for storing new code elsewhere.

9.1 NUMBER OF SECTORS PER GROUP

9.1.1 The smallest number of sectors that can be allocated to a file is one group, or 8 sectors. This is very wasteful, as few programs occupy more than two groups. The SD DOS allocates groups of 4 sectors at a time, with each sector containing half the number of bytes as DD sectors. This is not as profilgate as DD DOS, but still wastes space in the last group. One of the reasons for the group size is to reduce the size of the FAT. The FAT's of each disc have space reserved for them in RAM, and the bigger the FAT, the less user space there is available. The SD FAT size is 160 bytes, while 158 bytes are required for the DD DOS. If sectors were allocated individually, the DD FAT would need to be 1286 bytes long. It is probably because of this fact that we are stuck with groups of 8 sectors each.

9.1.2 There is room on the disc for an FAT to be 4 sectors long, even if the directory start remains at sector 5. This could easily be moved to make room for a larger FAT. However, the benefits to be gained by enlarging the FAT must be weighed against the fact that user RAM would be significantly reduced, and would result in a DOS that would be incompatable with the standard DOS.

9.2 DATE STAMPING OF FILES

9.2.1 Each entry in the directory occupies 32 bytes. Only the first 16 are used by the DOS, leaving the rest available for other uses. This space is ideal for storing file attributes such as the creation date,or time. It is used by HIWRITER for storing print attributes of files, such as margin and paragraph indentation.

9.2.2 The date stamping of files can be readily implemented by arranging the PEACH's date to be placed in the last 6 bytes of the directory entry whenever a file name is placed in the directory. A modification to the FILES code is then required to display the date beside each file name. A program to implement this modification has been included in APPENDIX III, and fits at the end of the DOS.

9.3 MULTIPLE-COLUMN DIRECTORY LISTING

9.3.1 An annoying feature of the DOS is that the FILES command displays in only one column, and very quickly scrolls off the screen. While the scrolling can be stopped be using CONTROL/S, the files already scrolled off can only be redisplayed by a new FILES command. The situation is not too hard to live with, with SD DOS, but with DD

discs holding many more files, the inconvenience is increased. The display can be improved by having the directory displayed in more than one column, making use of all of the screen, instead of only the left hand side.

9.3.2 When the FILES code has displayed one file name and its attributes, it displays a CRLF by a JSR $B0BE at $90E6. This call can be changed to a JMP to a program that shifts the current character position, not to the start of the next line, but to further along the same line. Only when the end of the line is reached is a new line taken. In this way, the standard directory can be displayed in 4 columns (80 character mode), or 2 columns (40 character mode).

9.3.3 If the DATE modification is implemented, each file name listed will take up 7 more bytes. Hence only 2 columns can be displayed in 80 character mode, and 1 column in 40 character mode. The program listed in APPENDIX III incorporates the DATE extension, and resides in the unused area at the end of the DOS.

9.4 RUNNING A PROGRAM ON BOOT UP

9.4.1 When the DOS initializes itself, it displays its sign-on message, then jumps to the same code (at $FD42) that the cassette system uses to display the number of free bytes. This jump can be redirected to code that will perform other functions, such as running a startup file, or loading extended commands.

9.4.2 Now the ROM code at $E6C5 loads the file name defined in $1FE-$205, (with $1FD being the number of characters in the file name), into memory and executes it. After initialization, the file name is NULL, i.e. it is filled with spaces, and $1FD=0. Hence a jump to $E6C5 after initialization will cause the NULL file to be run. This must be a BASIC file, but there are no restrictions in its use. The program in APPENDIX III incorporates this boot up procedure.

9.4.3 This procedure for running the NULL file seems to have something lacking as sometimes the screen controller chip is not set up properly. Some graphics programs will run with a funny screen mode. However, by holding down the BREAK key, the NULL file will load, but not run. The BREAK procedure resets everything correctly, so the program having difficulty may now be loaded and run properly.

9.4.4 If the DATE extension is used, it is a good idea to write a boot-up program to ask for the date to be entered from the keyboard, as in the following listing.

```
10 INPUT "Enter the date in the form (83/08/26)";A$
20 IF A$="" THEN DATE$="83/08/26" ELSE DATE$=A$
30 FILES
40 END
```

9.4.5 If the disc is to be used for games, the boot up file could be a menu of games to be selected by letter or number.

9.5 CHAINING PROGRAMS

9.5.1 When a program is loaded into RAM and executed, the ROM load-and-run system effectively performs a NEW command, thus clearing out any program that was in memory before the load. It is not possible to define variables in one program and use them in another program, without storing them first in a data file on disc. This process of loading in programs one after the other, and passing variables between them is called CHAINing.

9.5.2 A means of implementing chaining would be to move a program's symbol table to the top of RAM, load in the next program, then move the symbol table back down to just above the new program, its usual position. All the variables defined by the previous program would then be avaliable to the program just loaded.

9.5.3 This technique would require the LOAD command to be executed. This code is in ROM and returns to BASIC after completion. A return to a machine code program could be performed as explained in the notes for the OPEN command (section 4.1).

9.6 WILD CARD OPTIONS

9.6.1 It is sometimes desirable to perform a function several times on a number of different files. For example, display directory files that begin with FOX, or delete all files that have the extension TMP. By use of the wild card feature, a "don't care" character would be considered a match. In the following examples, the '*' is the wild card character.

```
FILES "1:FOX*.*"  List all files beginning with FOX..
KILL "1:*.TMP"    Delete all files with the extension TMP
```

9.6.2 This wild card feature could be coupled with a user query, so that each file name found to match would be displayed, along with a 'yes or no?' question.As you can imagine, enthusiastic use of the wild card could have disastrous effects, so a query acts as a second chance to detect mistakes.

9.6.3 The wild card capability comes under the category of nice, but not necessary. However, if extended commands are implemented as discussed below, the incorporation of a wild card feature would be feasible.

9.7 NEW COMMANDS

9.7.1 The 'PEACH ROM NOTES' gives a good account of how new commands can be added to the SD DOS. By following the instructions explained there, and with reference to the DD DOS commands discussed in these notes (section 7), new commands can be added to the DD DOS in a similar manner.

9.7.2 It would seem logical to put new commands in RAM just below the DOS, after the DOS has been initialized. The boot-up file could do this. Several BASIC pointers would need to be adjusted to point to the

new top of memory, the new end of string space, and the new stack region.

9.7.3 The pointers to be adjusted are:-

top of memory	$3D8, $2D
end of string space	$25, stack on return to BASIC

9.7.4 The space taken up by the new commands will eat into the user RAM, and some large programs such as STARTREK ADVENTURE will not fit into memory. If the NULL file is used to load the new commands, the user could be given the option to load them, or not.

9.8 DISC BASED COMMANDS

9.8.1 New commands added to the DOS will reduce the space available for programs, and could be quite a nuisance. If DOS commands could be stored on disc and only read into RAM when required, the space problem would be solved. It would be necessary to save the contents of the memory that the loaded command will occupy, and restore it when the command had finished its job. In this way, there is no limit to the sophistication of the new command, caused by lack of space.

9.8.2 The time to load the new command would need to be considered, particularly when DOS commands are always immediately available. Long routines would take a long time to load, and just as long to save RAM and restore it. Hence there is still pressure on the programmer to write compact code.

9.8.3 The question naturally arises as to where the command code would be stored on the disc, and how it would be loaded. One mechanism for storing these commands would be to produce them as relocatable binary files with the extension 'COM', and have a general command routine that would look for a command in the directory and load it. These COM files would need to be at the top of the directory, so that they would be found quickly. New commands could be added simply by putting the command file in the directory, and the command name in the command name table, once the general facility had been implemented. The nonexistant (to the DOS) DD sectors 17-32 on track 0 could be used as the scratch area (max of 4k), for saving the RAM area to be used by the commands.

9.8.4 Another means of storing the command code would be to have a table (referenced by command number) that contains the track,sector where the command code is stored on disc. These command programs would not be accessable via the directory, but have their own area set aside for them. These commands could be put on track 0, sectors 17-32, as the DOS doesn't know they are there. Thus no storage space would be taken from normal file storage on the disc. This scheme would have the advantage of quicker loading than the COM files, but it would not be so easy to add new commands. The suggested track 0 area would soon be filled, so commands would begin to take up file storage space also.

9.8.5 Some code would be needed in the DOS to recognize the new commands and then load the appropriate code from disc. Hence some space would be taken from user RAM for this command processing. BUT, some of the DOS commands are candidates for residing on the disc, so the new command processing could go in the space vacated by these DOS commands. These commands include DSKINI, FILES, NAME, VERIFY, DEVICE,and KILL. All of these commands are normally used in direct mode, and can be considered more like utilities, than program commands.

9.8.6 A considerable number of new commands come to mind, if the the space limitation were lifted. The following are a few suggestions, and I'm sure you could think of more.

a) List the directory in alphabetical order.
b) List only file names of the specified extension.
c) Short list the directory w/o date, 4 columns on screen.
d) Copy dev:file1 to dev:file2 with query.
e) Initialize a disc, and check for bad blocks. Put $FE in the FAT for the group in error.
f) Kill dev:file with query.
g) Remove REM's and spaces from a program.
h) Copy the display to the printer.

9.8.7 Because of the ease of implementing new commands, I tend to favour the COM files system for disc based commands, although the time to search for the command may be irritating. Directory searches could be restricted to (say) 2 sectors, i.e. the first 16 files in the directory to minimize directory search time.

9.8.8 Disc based commands thus would need to be processed in the following way.

a) Check that the token is valid.
b) Return to the token processing code at $D40D.
c) Each command processing routine would get the command name from the command name table, and call the general command routine.
d) The general routine would perform the following tasks.
   i) Write out, to a scratch area on disc, the RAM area to be used by the command.
   ii) Load the command into RAM and execute it. (When the command completes its task, control returns to the general routine).
   iii) Read the scratch area on disc back into RAM, and return to BASIC.

9.8.9 Two tables would also be needed as already explained for new commands in section 9.7.1. These would be the command names, and the command execution addresses.

10.0 FINAL THOUGHTS

10.0.1 I hope these notes have given you a better understanding of the internal workings of the DD DOS. I have tried to give enough information so that you can carry on your own investigations from a standpoint of knowledge, without having to rely on guess work.

10.0.2 I have also tried to give some ideas on how the DOS could be improved, without actually doing it. Perhaps someone who reads these notes will be inspired to take up the challenge and implement some of the suggestions. Most of these ideas are not original, and have been seen on other computer systems before. There is always a great deal of satisfaction, however, in putting some new feature into the DOS, particularly when others find it useful.

10.0.3 There is a danger in modifying the DOS that it becomes incompatible with standard DD DOS. THIS IS TO BE AVOIDED AT ALL COSTS. Any modified DOS should be able to read and write discs in standard form, otherwise CHAOS will result.

10.0.4 The serious DOS programmer will need three primary tools to help him in his task. A disassembler, an assembler, and a sector read/write/modify utility.These are readily available and make life much more enjoyable.

10.0.5 Well, I hope you have as much enjoyment in putting the DD DOS to good use, as I have had in producing these notes.


```
                *
                *DOUBLE DENSITY ROM BOOTSTRAP
                *READ TRACK 0,SECTOR 1,2 IN SINGLE DENSITY
                *
                *DISC REGISTERS
                *  FF00   READ=STATUS, WRITE=COMMAND
                *  FF01   CURRENT TRACK
                *  FF02   SECTOR
                *  FF03   DATA
                *  FF04   READ=DATA READY(MS BIT)
                *         WRITE=BIT 0,1=DRIVE
                *               BIT 3  =MOTOR ON
                *               BIT 4  =SIDE
                *               BIT 5  =DD=1, SD=0
                *               BIT 6  =NMI MASK
                *
                        ORG   $FE7E
                *
FE7E 12         ZFE7E   NOP                    ENTRY POINT
FE7F 12                 NOP
FE80 12                 NOP
FE81 CEFF00     ZFEB1   LDU   #$FF00           POINT TO REGISTERS
FE84 CC2F08             LDD   #$2F08
FE87 A7C4               STA   ,U               RESET CONTROLLER
FE89 E744               STB   4,U              DRIVE 0,SD,SIDE 0
FE8B 4F                 CLRA
FE8C 5F                 CLRB
FE8D 1F01               TFR   D,X              X=0
FE8F 8D0A       ZFE8F   BSR   ZFE9B            RTS ONLY
FE91 301F               LEAX  -$01,X
FE93 26FA               BNE   ZFE8F            WAIT 16 BIT COUNT
FE95 6DC4               TST   ,U               LOOK AT STATUS
FE97 2B03               BMI   ZFE9C            DRIVE PRESENT
FE99 6F44       ZFE99   CLR   $04,U            DRIVE NOT PRESENT
FE9B 39         ZFE9B   RTS                    RETURN TO CASS BASIC
                *
                *HERE AFTER CLEAR
                *
FE9C 4C         ZFE9C   INCA                   (A)=1
FE9D A5C4       ZFE9D   BITA  ,U               LOOK AT STATUS
FE9F 27FC               BEQ   ZFE9D            DRIVE BUSY
FEA1 867E               LDA   #$7E             SET UP NMI VECTOR
FEA3 B70109             STA   $0109
FEA6 308D09             LEAX  ZFEB2,PCR        TO FEB2
FEA9 BF010A             STX   $010A
FEAC 86FE               LDA   #$FE             RESTORE(01 INVERTED)
FEAE A7C4               STA   ,U
FEB0 20FE       ZFEB0   BRA   ZFEB0            WAIT FOR INT
                *
                *HERE AFTER RESTORE TO TRACK 0
                *
FEB2 326C       ZFEB2   LEAS  $0C,S            CLEAN UP STACK
FEB4 A6C4               LDA   ,U               GET STATUS
FEB6 8E4400             LDX   #$4400           LOAD BOOT BLOCK PROGRAM HERE
FEB9 5C         ZFEB9   INCB                   SECTOR 1
FEBA 3416               PSHS  X,B,A
```

```
FEBC 108E000A          LDY   #$000A       # TRIES ON ERROR
FEC0 8EFEDC    ZFEC0   LDX   #FEDC        SET UP INT RETURN AFTER SECT
FEC3 BF010A            STX   $010A
FEC6 ECE4              LDD   ,S           PUT SECTOR IN B
FEC8 AE62              LDX   $02,X        BUFFER ADR
FECA 53                COMB
FECB E742              STB   $02,U        SECTOR REGISTER
FECD 867F              LDA   #$7F         READ CMD (80 INVERTED)
FECF A7C4              STA   ,U           OUTPUT TO CMD REGISTER
FED1 A644      ZFED1   LDA   $04,U        LOOK AT MS BIT OF DRIVE REG
FED3 2BFC              BMI   ZFED1        NO CHARACTER AVAILABLE
FED5 E643              LDB   $03,U        READ CHARACTER
FED7 53                COMB
FED8 E780              STB   ,X+          STORE IN 4400+
FEDA 20F5              BRA   ZFED1        READ NEXT
                *
                *HERE AFTER READ OF SECTOR
                *
FEDC 8608      ZFEDC   LDA   #$08
FEDE A744              STA   $04,U        DRIVE 0,SD,SIDE 0,BIT 7=1
FEE0 326C              LEAS  $0C,S
FEE2 A6C4              LDA   ,U           GET STATUS
FEE4 43                COMA
FEE5 5F                CLRB
FEE6 85DC              BITA  #$DC         REMOVE UNWANTED BITS
FEE8 2705              BEQ   ZFEEF        GOOD READ
FEEA 313F              LEAY  -$01,Y       ERROR COUNT
FEEC 26D2              BNE   ZFEC0        TRY AGAIN
FEEE 43                COMA
FEEF 3526      ZFEEF   PULS  Y,B,A
FEF1 25A5              BCS   ZFE99        BAD READ, ABORT DISC
FEF3 C102              CMPB  #$02
FEF5 25C2              BCS   ZFEB9        READ SECOND SECTOR
FEF7 E61F              LDB   -$01,X       LOOK AT LAST CHAR READ IN
FEF9 C1DD              CMPB  #$DD
FEFB 269B              BNE   ZFE99        BAD, SHOULD BE DD
FEFD 7E4400            JMP   $4400        NOW RUN BOOT BLOCK PROGRAM
```



```
                *5 INCH DD BOOT BLOCK
                *  FOR HITACHI PEACH
                *  RESIDES ON TRACK 0,SECTOR 1 AND 2 OF DISC
                *
                *NOTE: DATA TO/FROM DISC REGISTERS IS INVERTED
                *
                *COMMENTS BY PETER CALDER,14 SANDY CRES,
                *              SALISBURY PARK,S. AUST
                *
                *83/04/26
                *
                *I/O REGISTERS
                *  FF00  STATUS(READ),COMMAND(WRITE)
                *  FF01  CURRENT TRACK
                *  FF02  REQD SECTOR
                *  FF03  DATA REG
                *  FF04  DRIVE REG  DATA READY(7), SD/DD(5), SIDE(4)
                *                   MOTOR ON(3),DRIVE(0-1)
                *
4400                    ORG   $4400
                *
4400 7E440B    Z4400    JMP   Z440B        ENTRY POINT
                *
4403 00        Z4403    FCB   $00          TRK #-START LOADING TRK 1
4404 20        Z4404    FCB   $20          SECTOR #
4405 8800      Z4405    FDB   $8800        DOS LOADS FROM HERE
4407 9FFF      Z4407    FDB   $9FFF        TO HERE
4409 FF        Z4409    FCB   $FF          CURRENT DRV REG VALUE
440A 06        Z440A    FCB   $06          # TRIES ON ERROR
                *
440B 8603      Z440B    LDA   #$03
440D B7FFC4             STA   $FFC4        RESET ACIA
4410 10CE45FF           LDS   #$45FF       INPUT BUFFER=4600+
4414 108EFF00           LDY   #$FF00       I/O REGS(FF00-FF04)
4418 BE4405             LDX   Z4405        PROGRAM LOAD ADR
441B 8606      Z441B    LDA   #$06
441D B7440A             STA   Z440A        6 TRIES ON ERROR
4420 3410               PSHS  X            X USED FOR LOADING INTO
4422 7C4404             INC   Z4404        SECTOR #
4425 F64404             LDB   Z4404
4428 C120               CMPB  #$20         END OF TRK?
442A 2323               BLS   Z444F        NO,READ NEXT SECTOR
442C 7C4403             INC   Z4403        YES,INC TRK #
442F 8601               LDA   #$01
4431 B74404             STA   Z4404        INITIALIZE TO SECTOR 1
4434 CE4446    Z4434    LDU   #Z4446       SET UP INT RET ADR
4437 FF010A             STU   $010A
443A B64403             LDA   Z4403        TRK #
443D 43                 COMA
443E A723               STA   $03,Y        IN FF03,DATA REG
4440 C6EE               LDB   #$EE         COMD 11,SEEK TRK
4442 E7A4               STB   ,Y           IN FF00,CMD REG
4444 20FE      Z4444    BRA   Z4444        WAIT FOR INT
                *
                *HERE AFTER SEEK TRK INTERRUPT
                *
```

```
4446 326C      Z4446  LEAS $0C,S        CLEAN UP STACK
4448 A6A4             LDA  ,Y           LOOK AT STATUS
444A 43               COMA
444B 8480             ANDA #$80
444D 2655             BNE  Z44A4        DRIVE NOT READY,ABORT
444F B64403    Z444F  LDA  Z4403        TRK #
4452 43               COMA
4453 A721             STA  $01,Y        IN FF01,CURRENT TRK REG
4455 8628             LDA  #$28         BITS=DD,MOTOR ON,DRIVE #
4457 F64404           LDB  Z4404        SECTOR #
445A C110             CMPB #$10         >16?
445C 2304             BLS  Z4462        NO
445E 8A10             ORA  #$10         YES,SET SIDE BIT
4460 C010             SUBB #$10
4462 A724      Z4462  STA  $04,Y        IN FF04, DRV REG
4464 B14409           CMPA Z4409        IS MOTOR ALREADY ON?
4467 270B             BEQ  Z4474        YES
4469 3410             PSHS X            NO,WAIT FOR MOTOR
446B 8E173E           LDX  #$173E
446E 301F      Z446E  LEAX -$01,X       COUNT BACK TO 0
4470 26FC             BNE  Z446E
4472 3510             PULS X
4474 53        Z4474  COMB
4475 E722             STB  $02,Y        IN FF02,SECTOR REG
4477 CE448C           LDU  #Z448C       SET UP INT RET ADR
447A FF010A           STU  $010A
447D C67F             LDB  #$7F         COMD $80,READ SINGLE SECTOR
447F E7A4             STB  ,Y           IN FF00,COMD REG
                      *
                      *READ DATA UNTIL END-OF-SECTOR INTERRUPT
                      *
4481 A624      Z4481  LDA  $04,Y        LOOK AT DATA READY BIT,FF04
4483 2BFC             BMI  Z4481        DATA NOT READY
4485 E623             LDB  $03,Y        FROM FF03,DATA REG
4487 53               COMB              DATA INVERTED
4488 E780             STB  ,X+          STORE DATA IN MEM
448A 20F5             BRA  Z4481        UNTIL INT AT END OF SECT
                      *
                      *HERE AFTER SECTOR READ IN
                      *
448C 326C      Z448C  LEAS $0C,S        CLEAN UP STACK
448E A6A4             LDA  ,Y           LOOK AT STATUS,FF00
4490 43               COMA
4491 849C             ANDA #$9C
4493 261C             BNE  Z44B1        BAD READ
4495 3540             PULS U
4497 BC4407           CMPX Z4407        END OF LOADING REGION?
449A 1023FF7D         LBLS Z441B        NO,READ NEXT SECTOR
449E 6F24             CLR  $04,Y        YES,CLEAR DRV REG,FF04
44A0 6E9F4405         JMP  [Z4405]      START DOS
                      *
                      *HERE IF DISC NOT READY AFTER SEEK
                      *OR TOO MANY ERRORS
                      *
44A4 6F24      Z44A4  CLR  $04,Y        CLEAR DRV REG,FF04
44A6 C639             LDB  #$39         PUT RTS INSTRUCTION
```



```
44A8 F70109           STB  $0109        IN NMI VECTOR ADR
44AB 10DE25           LDS  $0025        USE CAS BAS STACK
44AE 7EFB61           JMP  $FB61        START CAS BASIC
                      *
                      *HERE IF BAD READ
                      *
44B1 862F      Z44B1  LDA  #$2F         CMD DO,CLEAR
44B3 A7A4             STA  ,Y           IN FF00,COMD REG
44B5 8D1B             BSR  Z44D2        WAIT UNTIL NOT BUSY
44B7 CE44C3           LDU  #Z44C3       SET UP INT RET ADR
44BA FF010A           STU  $010A
44BD 86FE             LDA  #$FE         CMD 01,RESTORE
44BF A7A4             STA  ,Y           IN FF00,COMD REG
44C1 20FE      Z44C1  BRA  Z44C1        WAIT FOR INT
                      *
                      *HERE AFTER RESTORE INTERRUPT
                      *
44C3 326C      Z44C3  LEAS $0C,S        CLEAN UP STACK
44C5 8D0B             BSR  Z44D2        WAIT UNTIL NOT BUSY
44C7 A6E4             LDA  ,S
44C9 7A440A           DEC  Z440A        # TRIES ON ERROR
44CC 1026FF64         LBNE Z4434        TRY TO READ SECTOR AGAIN
44D0 20D2             BRA  Z44A4        TOO MANY ERRORS,ABORT
                      *
                      *WAIT FOR NOT BUSY
                      *OR TIMEOUT
                      *NOTE: STATUS IS INVERTED
                      *
44D2 CE0000    Z44D2  LDU  #$0000
44D5 8601      Z44D5  LDA  #$01         SET BIT 1
44D7 A5A4             BITA ,Y           LOOK AT STATUS,FF00
44D9 2604             BNE  Z44DF        WAIT FOR NOT BUSY
44DB 335F             LEAU -$01,U       COUNT UP TO 0
44DD 26F6             BNE  Z44D5
44DF 39        Z44DF  RTS               NOT BUSY OR TIMEOUT
                      *
44FF                  ORG  $44FF
                      *
44FF DD               FCB  $DD          ROM BOOT LOOKS FOR DD HERE
                      *
4400                  END  $4400
```

```
                * DSDD DOS MODIFICATIONS *
                *
                * 1.DISPLAY DATE IN DIRECTORY LISTING
                * 2.INSERT DATE IN DIR WHEN FILE CREATED
                * 3.RUN NULL FILE ON BOOT UP
                *
                * WRITTEN BY PETER CALDER
                *   14 SANDY CRESENT,SALISBURY PARK,S.AUST,5109
                * JAN 83
                * RUN NULL FILE OBTAINED FROM BRUCE ROSSELL,CANBERRA
                *
8A91                    ORG     $8A91
8A91 7E9FCE             JMP     RNUL            RUN NULL FILE
                *
90E6                    ORG     $90E6
90E6 BD9F6C             JSR     DATE            DATE DISPLAY
                *
945B                    ORG     $945B
945B BD9FBD             JSR     INSDT           INSERT TIME/DATE
                *
9F6C                    ORG     $9F6C
                *
                * DISPLAY DATE
                * REQUIRES JSR DATE AT 90E6
                *
9F6C 3436       DATE    PSHS    A,B,X,Y
9F6E B60238             LDA     $238            ALLOW FOR 2 DIGIT GROUP
9F71 8113               CMPA    #$13
9F73 2704               BEQ     D1
9F75 813B               CMPA    #$3B
9F77 2601               BNE     D2
9F79 4C         D1      INCA                    SPACE
9F7A 4C         D2      INCA
9F7B B70238             STA     $238
                *
9F7E B60116             LDA     $116
9F81 A7E2               STA     ,-S             SAVE COLOUR
9F83 8606               LDA     #$6
9F85 B70116             STA     $116            DATE IN YELLOW
                *
9F88 AE69               LDX     $9,S            POINT TO DIR ENTRY ADR
9F8A 30881A             LEAX    $1A,X           POINT TO DATE AREA
9F8D 8D25               BSR     OUT2            YEAR
9F8F 8D1F               BSR     OUSEP           MONTH
9F91 8D1D               BSR     OUSEP           DAY
                *
9F93 A6E0               LDA     ,S+
9F95 B70116             STA     $116            RESTORE COLOUR
                *
9F98 B60238     COL40   LDA     $238            <40?
9F9B 8128               CMPA    #$28
9F9D 240B               BHS     COLO            NO,START NEW LINE
9F9F 8628               LDA     #$28
9FA1 91A2               CMPA    $A2             40 COLUMN MODE?
9FA3 2705               BEQ     COLO            YES
9FA5 B70238             STA     $0238           NO, SET TO COL 40
```



```
9FA8 2003               BRA     RETN
                *
9FAA BDB0BE     COLO    JSR     $B0BE           PRINT CR
9FAD 3536       RETN    PULS    A,B,X,Y
9FAF 39                 RTS
                *
9FB0 862F       OUSEP   LDA     #$2F            '/'
9FB2 8D06               BSR     OUT             DISPLAY SEPARATOR
9FB4 8D00       OUT2    BSR     OUTC            DISPLAY 1st CHAR, THEN 2nd
9FB6 A680       OUTC    LDA     ,X+             GET CHAR
9FB8 8B30               ADDA    #$30            ADD ASCII BASE
9FBA 7EE820     OUT     JMP     $E820           DISPLAY CHAR
                *
                *INSERT DATE IN DIR
                * REQUIRES JSR AT $945B
                *
9FBD 3454       INSDT   PSHS    U,X,B
9FBF 334F               LEAU    $F,U            POINT TO TIME IN DIR ENTRY
9FC1 8E021C             LDX     #$21C           POINT TO TIME IN MEM
9FC4 C606               LDB     #$06
9FC6 BDFC48             JSR     $FC48           MOVE TIME/DATE
9FC9 3554               PULS    U,X,B
9FCB 7E9D6B             JMP     $9D6B           NOW WRITE SECTOR
                *
                * RUN NULL FILE *
                * REQUIRES JMP AT 8A91 AFTER INITIALIZATION
                *
9FCE DC25       RNUL    LDD     $25             LOW END OF STRING SPACE
9FD0 931D               SUBD    $1D             START OF PROGRAM SPACE
9FD2 BDC682             JSR     $C682           DISPLAY DIGITS
9FD5 8EFD9A             LDX     #$FD9A          "bytes free"
9FD8 BDB104             JSR     $B104           DISPLAY TEXT
9FDB 1CAF               ANDCC   #$AF
9FDD 7EE6C5             JMP     $E6C5           EXECUTE NULL FILE
                *
                        END
```